**Planner Configuration**



# Configuring the Built-in Planner with XML

This page describes how to use XML to configure EUROPA's built-in solver to solve a giving planning problem. We assume basic familiarity with the built-in solver and its terminology - see the Solver Description for details.

Note that the makeproject script includes a basic PlannerConfig.xml with the generated project to get you started.

## An Example

Suppose we want a planner to only make variable binding decisions, to exclude making any bindings on temporal variables, and variables of infinite or dynamic domains. This is a common requirement. Further assume that in general we want to pick the least value first, but in the case of setting the value for the speed of a drive token, we want to prefer to try choices in descending order. Finally, in the particular case of selecting values for *wind_speed* variables in *data* tokens on *Weather* object types, we wish to use a random selection with a uniform distribution and a cut-off which indicates how many of the possible values we want to try before admitting defeat. The XML snippet below illustrates how this might look.

```
<Solver name="TestSolver">
  <UnboundVariableManager defaultPriority="0">
    <FlawHandler component="StandardVariableHandler"/>

    <!-- Ignore temporal variables as well as vars wit infinite or dynamic domains -->
    <FlawFilter var-match="start"/>
    <FlawFilter var-match="end"/>
    <FlawFilter var-match="duration"/>
    <FlawFilter component="InfiniteDynamicFilter"/>

    <!-- Default behavior : try smallest value first -->
    <FlawHandler component="Min"/>

    <!-- for drive.speed try largest value first -->
    <FlawHandler predicate="drive" variable="speed" component="Max"/>

    <!-- for Weather.data.wind_speed try random with uniform(0,1) distribution, max 10 times -->
    <FlawHandler class="Weather" predicate="data" variable="wind_speed" component="Random" maxChoices="
  </UnboundVariableManager>
</Solver>
```

The key things to note are:

- A nested structure for components. The planner will be passed the XML element which in this case is the root. The planner will iterate through the child nodes. Components names of *UnboundVariableManager, ThreatManager, SAVHThreatManager, OpenConditionManager* will be used to obtain a concrete factory. Such factories will be passed in the xml element, which will include child nodes. The allowable data in the child nodes can be completely customized to each component.
- In this example, the planner is configured with only one *Flaw Manager*. This planner is thus only capable of making variable binding decisions. It will not address any *Object Flaws* or *Token Flaws*. No limit is placed on the number of *Flaw Managers* that may be configured.
- The order of definition of a *Flaw Manager* is the order in which they will be visited in the implementation of the function selectNextDecisionPoint.
- A *Resolver* binds to an instance of a *DecisionPoint* class. In the above example, resolvers are used for ordering and pruning. Such a role leaves the opportunity for incomplete search at the discretion of the application developer. Such details are encapsulated within a component.

# Basic Elements

The simplest Solver configuration that handles all of the basic types of flaw looks like this:

```
<Solver name="MySolver">
  <ThreatManager>
  </ThreatManager>
  <OpenConditionManager>
  </OpenConditionManager>
  <UnboundVariableManager>
  </UnboundVariableManager>
</Solver>
```

The "Solver" element simply provides a root for the XML and a name for the Solver, which is required. Underneath the Solver element are elements for the FlawManagers. These elements can either be the name of the FlawManager, as shown, or it can be a "FlawManager" element with a "component" attribute whose value is the name of the FlawManager. The following are equivalent:

```
<FlawManager component="UnboundVariableManager">
</FlawManager>

<UnboundVariableManager>
</UnboundVariableManager>
```

FlawManager elements can have a "defaultPriority" attribute that will assign that priority to any flaw of the appropriate type not matched by a FlawHandler. Multiple FlawManagers of a particular type are allowed, but will be evaluated in the order in which they appear in the text. This very important to note: in cases of equal priority, the Solver will select the flaw whose FlawManager appears earlier in the configuration. In the above example, though every flaw is given equal priority (no priority), threats will be handled first, followed by open conditions, then unbound variables. Additionally, FlawFilter elements may be placed under the Solver element, which will filter flaws before the FlawManagers get to look at them. For example, EUROPA provides a "HorizonCondition" which determines whether a flaw is within a particular temporal horizon that is given in the initial state:

```
<Solver>
  <FlawFilter component="HorizonFilter"/>
  <!-- then the FlawManagers -->
```

```
</Solver>
```

Underneath FlawManagers are FlawFilters and FlawHandlers. Despite being components, the convention where the name of the component is substituted for the element name, like with FlawManagers, cannot be used; the FlawManager uses the element name to determine whether something is a filter or a handler. FlawFilters and FlawManagers share a common matching facility for determining to which flaws they apply. The matching engine uses XML attributes to match "static" information, such as predicate names, and sub-elements for "dynamic" information, such as variable values.

Matching Attributes

| Attrubute | Synonym | Meaning | Values |
|---|---|---|---|
| class-match | class | Match the class of the object that the token is on | Any string |
| predicate-match | predicate | Match the predicate of the token | Any string |
| var-match | variable | Match the name of a parameter of the token | Any string |
| masterRelation | | Match the relationship between the token and its master. "none" matches the absence of a master | One of: before, after, meets, met_by, none |
| masterClass | | Match the class of the object that the master is on | Any string |
| masterPredicate | | Match the predicate of the master token | Any string |

The absence of any criterion is effectively a wild card, so a FlawFilter or FlawHandler without matching critera matches everything. The matching engine uses Guard and MasterGuard sub-elements to match the values of variables on the token or its master, both of which have a very simple name/value syntax. Here is an example of a relatively complex match, with some explanation:

```
<FlawHandler predicate="HGA_Comm" class="HGA_Class___HGA_Mode_SV" masterRelation="before" masterPredica
  <Guard name="dir" value="UPLINK"/>
  <MasterGuard name="dir" value="DOWNLINK"/>
</FlawHandler>
```

This defines a match for any token with the following properties:

- Predicate is `HGA_Comm`
- On an object of type `HGA_Class___HGA_Mode_SV`
- Has a parameter named "dir" whose value is `UPLINK`
- Required to be before its master
- Master has a predicate of `HGA_Comm`
- Master is on an object of `HGA_Class___HGA_Mode_SV`
- Master has a "dir" parameter whose value is `DOWNLINK`

Both FlawHandlers and FlawFilters can have a "component" attribute, but for different purposes. A FlawFilter component specifies additional matching criteria. A FlawHandler component usually specifies choice ordering criteria. Additional attributes and sub-elements are passed to these components and have per-component semantics.

The following sections describe the flaw managers, filters, and handlers that EUROPA provides, as well as any additional configuration they use or require.

# FlawManagers

These names can be used as the name of an element under a Solver element or as the value of a "component" attibute of a FlawManager element.

- OpenConditionManager: Handles open conditions. No additional configuration.
- ThreatManager: Handles threats for subclasses of Timeline. No additional configuration.
- SAVHThreatManager: Handles threats and decision ordering for subclasses of Unary, Reservoir and Reusable.
    - ♦ Additional configuration: an "order" attribute whose value is a comma-separated list of values determining the ordering criteria, with criteria to the left breaking ties for criteria on the left. The criteria are:
        - ◊ earliest/latest: Order flaws ascending/descending by the time at which they occur.
        - ◊ upper/lower: Prefer either upper- or lower-level flaws.
        - ◊ most/least: Order flaws ascending/descending by the distance between the level and the limit.
    - ♦ Example: `<SAVHThreatmanager order="lower,most,earliest"> <!-- prefer lower level, most flawed, temporally earliest flaws -->`
- UnboundVariableManager: Handles unbound variables. No additional configuration.

# FlawFilters

These names can be used as the value of a "component" attribute of a FlawFilter element.

## Filters on tokens

- MasterMustBeAssignedFilter: Filters out tokens whose masters have not yet been assigned. No additional configuration.
- HorizonFilter: Filters out tokens based on their start and end times relative to a horizon.
    - ♦ Additional configuration: a "policy" attribute whose value specifies the comparison with the planning horizon.
        - ◊ PossiblyContained: Filters out tokens whose start or end time does not interset the horizon.
        - ◊ PartiallyContained: Filters out tokens whose start and end times do not interset the horizon at all.
        - ◊ TotallyContained: Filters out tokens whose start or end time can possibly be outside the horizon.
    - ♦ Example: `<FlawFilter component="HorizonFilter" policy="PartiallyContained"/> <!-- Keep tokens that intersect the horizon -->`

## Filters on variables

- GuardFilter: Filters out guard variables. No additional configuration.
- NotGuardFilter: Filters out variables that aren't guards. No additional configuration.
- InfiniteDynamicFilter: Filters out variables with open or infinite domains. No additional configuration.
- SingletonFilter: Filters out variables until their derived domains are single values. No additional configuration.

- TokenMustBeAssignedFilter: Filters out variables on tokens that aren't active and whose object variable domains aren't single values. No additional configuration.
- HorizonVariableFilter: Exactly like the HorizonFilter, but filters variables based on the tokens to which they're attached. No additional configuration.

# FlawHandlers

FlawHandlers use matching criteria (including components) to associate a priority and decision point with a flaw. These names can be used as the value of a "component" attribute of a FlawHandler element.

## Variable handlers

- StandardVariableHandler: Selects values in ascending order. No additional configuration.
- Min: Synonym for StandardVariableHandler.
- Max: Selects values in descending order. No additional configuration.
- Random: Selects values in random order. No additional configuration.
- ValEnum: Selects values in a specified order.
    - ♦ Additional configuration: A series of "Value" sub-elements with "val" attributes specifying the allowed values.
    - ♦ Example:

```
<FlawHandler variable="foo" component="ValEnum"> <!-- Try "a", then "b", then "c" for var
    <Value val="a"/>
    <Value val="b"/>
    <Value val="c"/>
  </FlawHandler>
```

## Open condition handlers

- StandardOpenConditionHandler: Selects token states in the following order: MERGED, ACTIVE, REJECTED. Attempts to merge with active tokens in entity key order.
- HSTSOpenConditionDecisionPoint?: Allows a specification of preference for activation or merging and an order in which to select active tokens with which to merge.
    - ♦ Additional configuration:
        - ◊ "choice" attribute specifies the order in which to select token states with one of the following values (defaults to "mergeFirst"):
            - · activateFirst: Choose ACTIVE, then MERGED.
            - · mergeFirst: Choose MERGED, then ACTIVE.
            - · activateOnly: Only choose ACTIVE.
            - · mergeOnly: Only choose MERGED.
        - ◊ "order" attribute specifies the order in which to select active tokens with which to merge with one of the following values (defaults to "early", ascendingKey always used as a tie-breaker):
            - · early: order active tokens by ascending start time
            - · late: order active tokens by descending start time
            - · near: order active tokens by ascending temporal distance
            - · far: order active tokes by descending temporal distance
            - · ascendingKey: order active tokens by ascending entity key
    - ♦ Example: `<FlawHandler predicate="foo" component="HSTSOpenConditionDecisionPoint" choice="activateFirst"`

```
        order="near"/>
```

## Threat handlers

TODO! details

- StandardThreatHandler:
- SAVHThreatHandler:
- HSTSThreatDecisionPoint:
- ResourceThreatDecisionPoint: